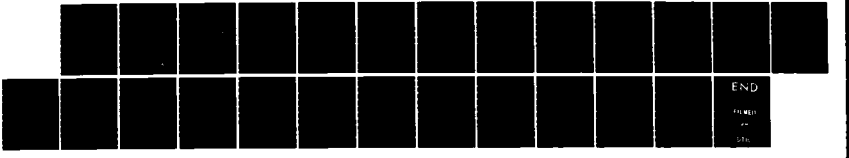
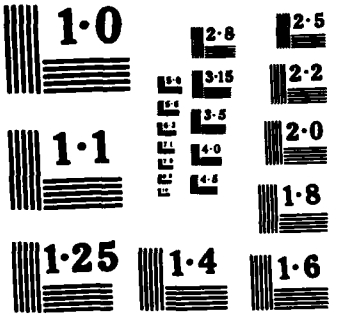
AD-A162 954 UPDATING PROPERTIES OF DIRECTED ACYCLIC GRAPHS ON A PARALLEL RANDOM ACCES..(U) MARYLAND UNIV COLLEGE PARK CENTER FOR AUTOMATION RESEARCH S PAWAGI ET AL. SEP 85 1/1

UNCLASSIFIED CAR-TR-148 AFOSR-TR-85-1124 F/G 9/2 NL





CAR-TR-148
CS-TR-1551

September 1985

Updating Properties of Directed Acyclic Graphs on a Parallel Random Access Machine +

Shaunak Pawagi
I. V. Ramakrishnan †

Department of Computer Science
University of Maryland
College Park, MD 20742

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

CAR–TR–148
CS–TR–1551

September 1985

Updating Properties of Directed Acyclic Graphs on a Parallel Random Access Machine [+]

Shaunak Pawagi
I. V. Ramakrishnan [†]

Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Fast parallel algorithms are presented for updating the transitive closure, the dominator tree, and a topological ordering of a directed acyclic graph (DAG) when an incremental change has been made to it. The kinds of changes that are considered here include insertion of a vertex or insertion and deletion of an edge. The machine model used is a parallel random access machine which allows simultaneous reads but prohibits simultaneous writes into the same memory location. The algorithms described in this paper require O(log n) time and use $O(n^3)$ processors. These algorithms are efficient when compared to previously known $O(\log^2 n)$ time algorithms for initial computation of the above mentioned properties of DAGs. We also describe a new algorithm for initial computation of the dominator tree of a DAG. Our algorithm improves the processor complexity of a previously known algorithm [14] by a factor of n, but does not affect the time complexity, which remains $O(\log^2 n)$.

F49620-83-C-0082

[+] The support of the first author by the Air Force Office of Scientific Research under Contract F-49620-85-K-0009, and of the second author by the Office of Naval Research under Contract N00014-84-K-0530, and by the National Science Foundation under grant ECS-84-04399, is gratefully acknowledged.

[†] Present address: Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794.

1. Introduction

Incremental graph algorithms are concerned with recomputing properties of a graph after a minor modification has been made to the graph. Such recomputations are also referred to as "updating" graph properties. Sequential incremental algorithms for recomputing minimum spanning trees [17], connected components [5], transitive closure [9], and shortest paths [6] have appeared in the past. The kinds of minor changes that are considered in incremental computations are as follows. First, a vertex may be added along with the edges incident on it. Second, an individual edge joining two vertices may be deleted or added. If edges have weights associated with them then an increase or decrease in the weight of an individual edge is a minor change. For such minor modifications it should be possible to design efficient algorithms for recomputing properties of graphs when compared to the start-over algorithms [3] that do not assume existence of the previous solution.

We can characterize incremental algorithms in terms of stages. The first stage is to determine what part of the solution is unaffected by the graph change. This is important as substantial gains can be made by avoiding the recomputation of the unaffected part of the solution. The second stage is the actual recomputation of that part of the solution which is affected by the graph change. This stage can be implemented efficiently by using the previous solution and possibly some auxiliary information that is generated during the initial computation of the solution. This in turn leads us to a third stage which consists of updating the auxiliary information. The complexity of an incremental algorithm depends on the complexity of these three stages and our objective is to design incremental algorithms that are efficient when compared to start-over algorithms.

Pawagi and Ramakrishnan were the first to treat the problem of incremental computations in graph theory in the context of synchronous parallel computation. They have described efficient algorithms for updating minimum spanning trees [11], connected components and bridges [13], and the distance matrix and shortest paths [12] of an undirected graph on an unbounded model of a parallel random access machine (PRAM). In this model of computation all processors have access to a global memory and processors can simultaneously read from the same location but no two processors can simultaneously write into the same location. We refer to this model as R-PRAM. Parallel algorithms for several graph problems have been devised on this particular model of computation [4, 7, 15, 18]. The algorithms developed on this model provide us with a basis for comparing the complexity of our incremental algorithms. In this paper we describe incremental algorithms for updating the transitive closure, the dominator tree and a topological ordering of a DAG on an R-PRAM. We consider the above mentioned minor changes except for a change in the weight of an edge. Our algorithms for updating these properties require $O(\log n)^{++}$ time and therefore are efficient when compared to the start-over algorithms for initial computation of these properties that require $O(\log^2 n)$ time.

A powerful variation of a PRAM is a model that has a concurrent write feature. We refer to this model as W-PRAM in which more than one processor can simultaneously write different values into the same memory location and only one processor succeeds but we do not know which. Start-over algorithms for initial computation of the above mentioned properties require O(log n) time and use $O(n^4)$ processors on this model [10]. An important feature of our algorithms for updating the transitive closure and topological ordering is their versatility, that is, they can be adapted to run on a W-PRAM using $O(n^4)$ processors. Our incremental algorithms require O(1) time on a W-PRAM write

$^{++}$Throughout this paper, we use log n to denote $\lceil \log_2 n \rceil$

model and are thus efficient when compared to start-over algorithms.

The rest of the paper is organized as follows. In Section 2 we describe some graph theoretic preliminaries. In Section 3 we describe our algorithms for updating the transitive closure. In Section 4 we present our new start-over algorithm for computing the dominator tree. The incremental algorithm for updating the dominator tree is also described in this section. In Section 5 we extend the ideas to incremental computation of a topological ordering. Concluding remarks appear in Section 6 where we discuss the adaptability of our incremental algorithms.

2. Preliminaries

In order to describe our algorithms for updating properties of a DAG we now define some graph theoretic terms and explain our notation for them.

Let G=(V,E) denote a *graph* where V is a finite set of vertices (nodes) and E is a set of pairs of vertices called edges. If the edges are unordered pairs then G is *undirected* else it is *directed.* Throughout this paper we assume that V consists of the set of vertices {1,2,...,n} and |E|=m. We denote the undirected edge from u to v by (u,v) and the directed edge joining them by $<u,v>$. An adjacency matrix A of G is an $n \times n$ Boolean matrix such that A[u,v]=1 if and only if (u,v) is in E. A path in G joining two vertices i_0 and i_k is defined as a sequence of vertices $(i_0,i_1,i_2...,i_k)$ such that all of them are distinct and for each $0 \leq p < k$, (i_p,i_{p+1}) is an edge of G. If $i_0 = i_k$ then the path is called a cycle. We denote an undirected path from vertex u to vertex v by [u-v] and a directed path by [u→v]. We say that an undirected graph G is *connected* if for every pair of vertices u and v in V, there is a path in G joining u and v. Each connected maximal subgraph of G is called a *component* of G. A *tree* is a connected undirected graph with no cycles in it. A *rooted* directed tree has a distinguished vertex called the

root, from which every other vertex is reachable via a directed path. We say that vertex u is an *ancestor* of vertex v if u is on the path from the root to v. A *descendant* of a vertex is defined similarly. The *lowest common ancestor* (LCA) of vertices x and y in T is the vertex z such that z is a common ancestor of x and y, and any other common ancestor of x and y in T is also an ancestor of z in T.

3. Transitive Closure

It has been shown in [18] that several graph properties of an undirected graph can be computed by first constructing a spanning subtree for the graph. Consequently, update algorithms for these properties involve updating a spanning tree for the new graph (see [11, 13]). Similarly, start-over algorithms for initial computation of properties of a DAG require its transitive closure to be computed. Therefore the update of the transitive closure of a DAG is an important step in incremental algorithms for updating properties of a DAG. In this section we describe our algorithms for updating the transitive closure of a DAG after an edge has been inserted or deleted from it or a vertex has been inserted into it.

Definition: The transitive closure of a directed graph G is an $n \times n$ boolean matrix A^* such that $A^*[i,j] = 1$, iff there is a directed path from i to j, otherwise $A^*[i,j]$ is 0.

Note that for an acyclic graph, $A^*[i,i]$ must be 0, for all i.

The problem of updating the transitive closure involves recomputing A^* for the modified graph. Our algorithms for updating the transitive closure on an R-PRAM require O(log n) time and use $O(n^3)$ processors. The start-over algorithm for initial computation of A^* requires $O(\log^2 n)$ time and uses $O(n^3)$ processors [7]. Our algorithms therefore are efficient when compared to the start-over algorithm. To design efficient parallel

algorithms for updating the transitive closure we proceed as follows.

Instead of computing the boolean matrix A^*, we compute the lengths of the shortest paths for all vertex pairs and store them in A^*. This computation assumes that edges have unit weights. Now $A^*[i,j]$ is the length of the directed shortest path from i to j. We refer to the length of a shortest path $[i \rightarrow j]$ as the distance from i to j. The first step in recomputing A^* is to determine the vertex pairs whose distances are unaffected by the graph change. In particular, we need to compute these pairs after an edge has been deleted from G. The other cases of edge and vertex insertion are easy to handle. We do not consider the problem of vertex deletion, because we are unable to determine the vertex pairs whose distances remain unchanged after vertex deletion.

In order to describe the actual computational steps of our algorithms and the proof of their correctness, we first describe the parallel start-over algorithm for computation of the transitive closure.

Start-Over Algorithm

It has been observed in [4] that distances for all pairs of vertices in a graph (directed or undirected) can be computed in $O(\log^2 n)$ time on a R-PRAM by straightforward parallelization of the known sequential algorithm that is based on repeated multiplication of the adjacency matrix. In this parallel algorithm addition and minimization operations replace the multiplication and addition operations of an inner product step involved in ordinary matrix multiplication. We refer to this as the *plus-min* multiplication of two matrices. The algorithm initializes the transitive closure A^* to the adjacency matrix A and then performs log n iterations of the plus-min multiplication of A^* by itself. The matrix DD is used as temporary storage for clarity.

// All steps involving i and j are executed for all i, j $1 \leq i \leq n$ and $1 \leq j \leq n$ //

1. $A^*[i,j] := A[i,j]$ //Initialize//
2. for t:=1 to log(n-1) do
2a. $DD[i,j] := \min \{ A^*[i,j], A^*[i,k] + A^*[k,j] \}$ // $1 \leq k \leq n$ $i \neq k$ $j \neq k$//
2b. $A^*[i,j] := DD[i,j]$

Algorithm 3.1

Lemma 3.1: The above algorithm computes the transitive closure A^* in $O(\log^2 n)$ time using $O(n^3)$ processors.

Proof: Steps (1) and (2b) can be done in constant time using n^2 processors. Step (2a) can be done in O(log n) time by assigning n processors to compute each element of the matrix DD. Since DD has n^2 elements we need $O(n^3)$ processors to perform step (2a). Note that at the end of t^{th} iteration we would have found distances for those pairs whose vertices are at most 2^t units apart. Since the maximum distance for any pair of vertices is at most n-1 units, we need log(n-1) iterations of step (2a).

We denote the processor complexity of Algorithm 3.1 by $P_{tc}(n)$. It can be easily improved to $O(n^3/\log n)$ using a technique described in [8]. If Chandra's [2] algorithm for matrix multiplication is used in step (2a) then the processor complexity reduces to $O(n^{2.81}/\log n)$. We now proceed to describe our update algorithms and to prove their correctness.

Edge deletion

The problem of edge deletion update is concerned with recomputing the transitive closure A^* after an edge has been deleted from the graph. In order to recompute A^*, we first identify the pairs of vertices whose distances are unaffected by the edge deletion step. We then construct matrix A_u^* (u stands for unaffected) such that

$$A_u^*[i,j] = \begin{cases} A^*[i,j], & \text{if } A^*[i,j] \text{ is unchanged.} \\ \infty & \text{otherwise} \end{cases}$$

Now, two iterations of steps (2a) and (2b) of Algorithm 3.1 on A_u^* recompute the transitive closure for the new graph. We will show later on that two iterations are sufficient for recomputing A^*.

Let $<x,y>$ be the deleted edge. Note that the i^{th} row of A^* corresponds to a shortest path tree that is rooted at r. Deletion of an edge from G may disconnect these trees affecting the distances from the root to vertices in the subtrees that are now rooted at y (see Fig 3.1). For i^{th} row we want to determine the vertices whose distances from i might have been affected by deletion of $<x,y>$. The computational steps given below are for the i^{th} row, but are executed for all rows in parallel. Let d_j denote the distance to vertex j from the root i.

1. If $d_j = d_y + A^*[y,j]$, then deletion of $<x,y>$ may affect d_j. Therefore for all such j, set $A_u^*[i,j] = \infty$. All other entries in the i^{th} row are not affected. As there are $O(n^2)$ entries to check we need $O(n^2)$ processors.
2. Perform two iterations of the start-over algorithm on A_u^* to compute the updated transitive closure matrix. This computation requires O(log n) time and uses $O(n^3)$ processors.

We now prove the correctness of our algorithm.

Lemma 3.2: Two iterations of the start-over algorithm that operates on A_u^* are sufficient to compute the updated transitive closure for the new graph.

Proof: Consider the shortest path tree S_i corresponding to the i^{th} row that is rooted at i (see Fig. 3.1). Let $<x,y>$ be the edge that was deleted. Deletion of $<x,y>$ creates two

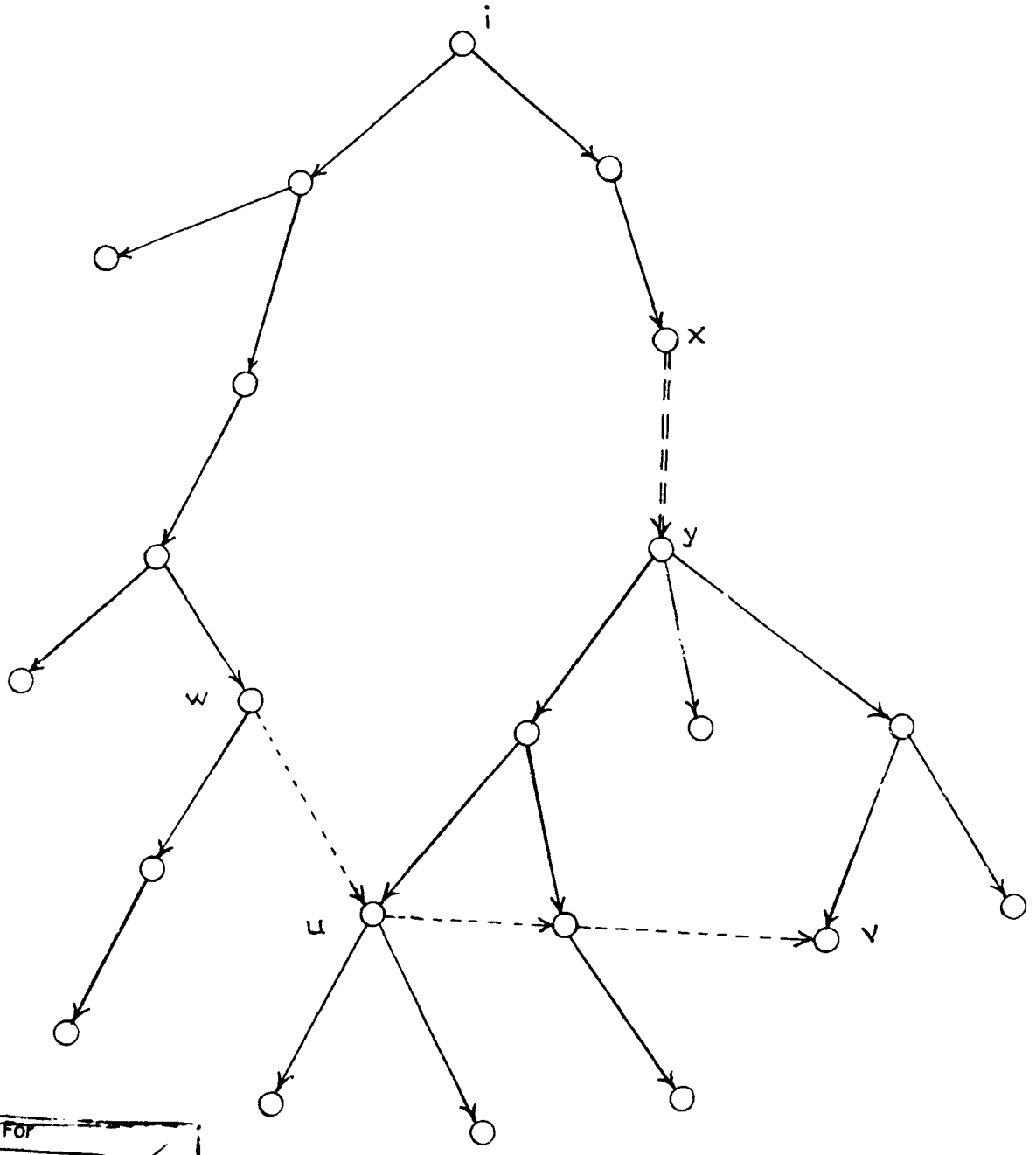


Fig. 3.1

subtrees, one rooted at i and the other rooted at y. If $<x,y>$ is an edge of G such that all paths from i to a descendant of y, say v, in S_i use $<x,y>$, then after deletion of $<x,y>$, v is not reachable from i in G. Therefore the distances from i to all such vertices v are set ∞ in A_u^*, and stay ∞ even at the end of the second iteration of the start-over algorithm that operates on A_u^*.

On the other hand if $<x,y>$ is not such an edge for v then there exists another path from i to v that does not use the edge $<x,y>$. For the purpose of analysis, assume without loss of generality that such a path $[i \rightarrow v]$ consists of three subpaths, namely, one from i to w, one from w to u, and one from u to v. These paths have the following characteristics: (i) the distance from i to w is not affected by deletion of $<x,y>$, (ii) either $<w,u>$ is an edge of G, or u is equal to w, (iii) the path $[u \rightarrow v]$ uses only vertices that are descendants of y in S_i, and its length is not affected by deletion of $<x,y>$.

It is easy to see that if there exists another path from i to v then it can be always split into three subpaths as mentioned above. Vertex w is the last vertex from i on the path $[i \rightarrow v]$ whose distance from i is unaffected by deletion of $<x,y>$. The path $[w \rightarrow u]$ is either a single edge or possibly null and cannot possibly use $<x,y>$. The path $[u \rightarrow v]$ uses all descendants of y and cannot use $<x,y>$ because G is acyclic.

In fact there might be more than one such u from which v is reachable. Therefore at the end of the first iteration (i.e., *plus-min* multiplication of A_u^* by itself) we would have found the distances to all such u's from i.

Now consider a vertex v that does not have a neighbor in the subtree rooted at i. The shortest path from i to v must pass through some such u (a vertex that has a neighbor in the subtree rooted at i). Therefore the distance to v from i can be expressed as the sum of two distances, one from i to u and the other from u to v. Now, at the end of

the second iteration we would have computed the distances to all such v's from i. This requires that $A_u^*[u,v]$ must not have been marked ∞ in step (1) of our algorithm. In other words, v must be reachable from u even after the edge $<x,y>$ has been deleted from G. This is always true as the path $[u \rightarrow v]$ cannot use $<x,y>$.

It is possible that vertex v is reachable from many such u's, but the minimization operation will select a vertex that minimizes the length of the path from i to v. Therefore at the end of the second iteration we would have computed the shortest paths to all vertices from the root i. Hence the theorem.

Theorem 3.1: Our algorithm updates the transitive closure of a directed acyclic graph after an edge deletion operation in O(log n) time and uses $O(n^3)$ processors.

Proof: The proof is immediate from steps 1 and 2 of our algorithm.

Edge and Vertex Insertion

We now describe our algorithms for updating the transitive closure matrix after an edge or a vertex has been inserted into G. In order to compute A_u^* from A^* after an edge insertion operation we proceed as follows. Let $<u,v>$ be the edge that has been inserted into G.

1. Set $A_u^*[u,v] = 1$. All other entries of A_u^* are the same as those of A^*.
2. Perform two iterations of the start-over algorithm that uses A_u^* as its input to compute the updated transitive closure.

In the case of vertex insertion we add a new row and a column to the old transitive closure. Let z be the new vertex that has been inserted into G. Now A_u^* can be obtained from A^* by setting $A_u^*[z,w] = 1$, for all w, where $<z,w>$ is an edge, and $A_u^*[y,z] = 1$,

for all y, where $<y,z>$ is an edge. All the other entries in the z^{th} row and in the z^{th} column are ∞. Again, two iterations of the start-over algorithm recompute the new transitive closure.

Theorem 3.2: Our algorithms for edge and vertex insertion update require O(log n) time and use $O(n^3)$ processors.

Proof: For an edge insertion update we can compute A_u^* from A^* in constant time using one processor and this step for a vertex insertion update requires 2n processors. The rest of this proof is along lines similar to that of Theorem 3.1.

4. Dominators

In this section we describe our start-over algorithm for computing the dominator tree of a DAG and an incremental algorithm for updating it. Computing the dominators of a directed acyclic graph (DAG) is a very important code optimization step in compilers (see [1] for details).

A directed graph is *rooted* at r if there is a path from r to every vertex in V. For the rest of this section, without loss of generality we shall assume that G is a directed acyclic graph rooted at r. Vertex i is a *dominator* of vertex j if i is on every path from r to j. In particular, for every i in V, r and i are dominators of i. Dominators exhibit transitivity, that is, for vertices i,j and k in V, whenever i is a dominator of j and j is a dominator of k, then i is a dominator of k. Therefore it is easy to see that the set of dominators of a vertex j can be linearly ordered by their order of occurrence on a shortest path from r to j. The dominator of j closest to j (other than j) is called the *immediate* dominator of j. Observe that the immediate dominator of every vertex is unique. We can now express the dominator relation as a directed tree D_t rooted at r called the

dominator tree. If u is the immediate dominator of v then $<u,v>$ is an edge of D_t. Now i is a dominator of j if i is an ancestor of j in D_t.

For each vertex k, Savage [14] first constructs a graph G_k, by deleting from G all edges leaving k. Next, the transitive closure of each of these graphs is computed in parallel. Let G^* and G_k^* denote the transitive closures of G and G_k respectively. Dominators are then computed by using the simple observation that if i is reachable from r in G^* and not in G_k^*, then k is a dominator for i. Savage's algorithm requires $O(\log^2 n)$ time and uses $O(nP_{tc}(n))$ processors, where $P_{tc}(n)$ is the processor complexity of computing the transitive closure.

In the case of undirected graphs, the processor complexities of computing a minimum spanning tree, bridges, bridge-connected components, cut points, and biconnected components are same as that of computing connected components (see [18]). Analogously, the processor requirements for computing properties of a DAG should be determined by the processor complexity of computing the transitive closure of a directed graph. In the following section we will describe our algorithm for computing the dominators on an R-PRAM. Our algorithm has $O(\log^2 n)$ time complexity and uses $O(P_{tc}(n))$ processors. Observe that our algorithm requires fewer processors by a factor of n than Savage's algorithm, no matter what algorithm is used for computing the transitive closure.

Start-over Algorithm

In order to compute the dominator tree we first construct a shortest path tree for G that is rooted at r. We then compute the set of dominators for each vertex in a matrix DOM such that DOM[i,j] = 1 if i is a dominator of j, otherwise DOM[i,j] = 0. The computational steps are as follows.

1. Compute the transitive closure matrix A^* for G. By Lemma 2.1, this computation requires $O(\log^2 n)$ time and uses $O(P_{tc}(n))$ processors.

2. Compute the shortest path tree S_t from the adjacency matrix A and the transitive closure matrix A^*. This is done by specifying the father of each vertex. Let vertex i be at distance d from r. A parent vertex for i is a vertex j such that the distance to j from r is d-1 and $<j,i>$ is an edge of G. Break the ties by selecting the minimum such j. This step requires $O(\log n)$ time and uses $O(n^2)$ processors.

3. For every vertex, mark all its ancestors in S_t as its dominators. That is, set DOM[i,j] to 1 if i is an ancestor of j, else DOM[i,j] is set to 0. Ancestor computation can be done in $O(\log n)$ time using $O(n^2)$ processors (see [18]). Initialization of the matrix DOM requires constant time and $O(n^2)$ processors.

4. For every vertex v, consider the non-tree edges incident on it. For all such edges $<x,v>$, compute the lowest common ancestor w of x and v in S_t. Select a non-tree edge $<u,v>$ such that $w_h = LCA(u,v)$ in S_t, is closest to the root r (h stands for the highest). The lowest common ancestors for all vertex pairs can be computed in $O(\log n)$ time using $O(n^2)$ processors (see [18]). For each vertex v, w_h and $<u,v>$ can be determined in $O(\log n)$ time using n processors for each vertex.

5. Now, the edge $<u,v>$ provides a path from w_h to v passing through u, other than the path present in S_t. Therefore all vertices on the path $[w_h \rightarrow v]$ are not dominators of v. For all such vertices j, set DOM[j,v] = 0. Since the number of vertices on any path is at most n, we need $O(n^2)$ processors to do this step in constant time.

6. For every vertex i, if j is not a dominator of i then j is not a dominator of any vertex reachable from i. Set DOM[j,x] = 0 for all x reachable from i. This is equivalent to *plus-min* multiplication of two matrices, DOM and A^*, because A^*

contains the reachability set for each vertex i. Therefore this step requires O(log n) time and $O(P_{tc}(n))$ processors.

This completes the description of our algorithm for dominators. We now provide the proof of its correctness.

Lemma 4.1: If vertex u is not a dominator of vertex v then at the end of our algorithm DOM[u,v] will be set to 0.

Proof: If u is not on a shortest path from r to v then DOM[u,v] is set to 0 in step 2 of our algorithm and it stays 0 for all the following steps. If u is on a path from r to v but it is not a dominator of v then there must exist a path from r to v that does not pass through u. There are two cases to be considered. First, v has a non-tree edge <x,v> incident on it such that u is on a path from LCA(x,v) to v. Among the LCAs for all non-tree edges <x,v> incident on v, step (5) selects w_h such that it is closest to the root. In this case, step 5 of our algorithm will set DOM[u,v] = 0, because u must lie on the directed path from w_h to v. Second, v is reachable from vertex y such that y has a non-tree edge incident on it, providing another path to y from r, that does not use u. In step 5 of our algorithm DOM[u,y] will be set to 0, and since v is reachable from y, in the next step, DOM[u,v] will be set to 0. Hence the Lemma.

Theorem 3.1: The above algorithm computes the dominator matrix DOM in $O(\log^2 n)$ time using $O(P_{tc}(n))$ processors.

Proof: The correctness of our algorithm is proved in Lemma 3.1 and the processor and time complexities are immediate from steps 1 to 6 of our algorithm.

Given the matrix DOM, and the shortest path matrix A^*, the dominator tree can be easily constructed. Recall that the immediate dominator of a vertex is unique and it is

the closest dominator of that vertex. Assign n processors to each vertex and select a dominator that is closest to it. Such a selection can be done in O(log n) time because it involves computing a minimum of at most n elements. The root of this tree is r and the father of each vertex except r is its immediate dominator.

This completes the description of our start-over algorithm for computation of the dominator tree.

Updating the Dominator Tree

Observe that all steps, except the first, of our start-over algorithm for computing the dominator tree require O(log n) time. The first steps requires $O(\log^2 n)$ time since it involves computing the transitive closure for G. In Section 3 we have described our incremental algorithms for updating the transitive closure of a DAG, after an incremental change has been made to it. Instead of using a standard boolean matrix for transitive closure we used the shortest path matrix to represent the reachability set for each vertex. Since we are able to update the modified transitive closure matrix in O(log n) time we can update the dominator matrix for G in O(log n) time using $O(n^3)$ processors. Construction of dominator tree from the dominator matrix requires O(log n) time. Therefore we have the following theorem.

Theorem 4.2: Given the transitive closure as shortest path matrix for the original DAG G, we can update the dominator matrix and the dominator tree for G after an incremental change has been made to it in O(log n) time using $O(n^3)$ processors.

5. Topological Ordering

In this section we describe algorithm for updating a topological ordering of vertices in a DAG. This is an important property of DAGs and finds applications in activity

networks and in critical path analysis of networks.

Dekel et al. [4] observed that a topological ordering can be computed using the longest paths for all vertex pairs. Kucera [10] has described a simple algorithm for topological ordering of vertices which makes use of the following definition. For different vertices u and v of a DAG G, u is a predecessor of v iff there is a directed path from u to v. The topological ordering of the vertices has the property that if vertex i is a predecessor of vertex j, then i precedes j in a topological ordering of the vertices of G. The steps are given below.

1. Compute the transitive closure of the given directed acyclic graph G. This computation requires $O(\log^2 n)$ time and uses $O(n^3)$ processors.
2. For every vertex j, determine the cardinality C_j of the set of vertices from which j is reachable. This can be done in O(log n) time by assigning n processors to each vertex as it involves computing a sum of n elements.
3. To obtain the topological ordering sort the vertices using C_j as a key.

Lemma 5.1: The above algorithm computes a topological order of the vertices.

Proof: By the definition of topological order, if vertex i is a predecessor of vertex j then i occurs before j in the ordering. Now C_i is less than C_j because all predecessors of i are also predecessors of j. Therefore when all vertices are sorted on C_k, $1 \leq k \leq n$, i must occur before j in the topological ordering.

The time and processor complexities of our algorithm are $O(\log^2 n)$ and $O(P_{tc}(n))$ respectively.

Updating a Topological Ordering

It easy to see that we can update topological ordering if we can update the transitive closure of a DAG in O(log n) time. By Theorem 3.1, the transitive closure of a DAG can be updated in O(log n) time using $O(n^3)$ processors, and the sorting step needs O(log n) time. Therefore we can update a topological ordering in O(log n) time. In order to come up with incremental algorithms that work for all variations of PRAM we have to avoid the sorting step during updating of a topological ordering, because the sorting step cannot be done in O(1) time on a W-PRAM.

We now proceed to describe the incremental algorithms for updating a topological ordering that can be adapted to run on a W-PRAM. Assume that the topological ordering is stored in an array T_o such that $T_o(i)$ give the numerical ordering of i.

Among the three graph changes considered in this paper, deletion of an edge from G does not affect a topological ordering. Deletion of a vertex requires reducing the rank of the subsequent vertices in T_o by 1. This can be done in constant time using n processors. Finally, as observed by Cheston [3], addition of a vertex is equivalent to adding an edge. Therefore we concentrate on edge insertion update. The steps involved are as follows. Let $<u,v>$ be the edge that has been added to G.

1. If $T_o(u) \leq T_o(v)$ then the previous ordering requires no update, else proceed with step 2.

2. Since $T_o(u) > T_o(v)$ it is sufficient to move v and those of its successors that appear in the topological order before u to positions after u.

   This step is done using a known algorithm for merging two sorted lists, both of size n [16]. Next, create a new list of vertices that consists of v and its successors that appear in the topological ordering before u. The order of the vertices is the

same as before. The vertices that are now in the new list are removed from the old one. Now increment the rank of each vertex in the new list by n and that of all vertices after u in the old list by 2n. We now have two sorted lists of size at most n. These lists are then merged using a known algorithm. This can be done in constant time using $O(n^{1.5})$ processors [16]. The new rank of each vertex is its position in the array containing the merged list.

Now consider the insertion of a new vertex w that has several incident edges. Let v be the topologically first node that has an edge <w,v> incident on it. Then the ranks of all vertices that precede v are unchanged. Insert w in the list just before v. Now we have n+1 vertices in the list. Therefore we need to increment by 1 the rankings of v and all vertices that appear after v in the old order. Let u be the topologically last node that has an edge <u,w> incident on w. Therefore we need to move w and its successors in the new graph G, to positions immediately following u. This is equivalent to an insertion of edge <u,w> and can be done in constant time using the edge insertion algorithm. The above discussion is summarized in the following lemma.

Lemma 5.2: A topological ordering of a DAG can be updated in O(log n) time using $O(n^3)$ processors.

Proof: All steps except updating of the transitive closure can be done in constant time and with $O(n^{1.5})$ processors. Therefore the time and processor complexities are determined by the first step which involves updating of the transitive closure. By Theorem 3.1, this can be done in O(log n) time using $O(n^3)$ processors. Hence the theorem.

6. Conclusions

In this paper we have presented a set of algorithms to update properties of directed acyclic graphs such as transitive closure, dominators and topological ordering on an R-PRAM. The central idea was to update the transitive closure of a DAG that has been modified to store the lengths of the shortest paths for all vertex pairs.

An important feature of our algorithms (except the incremental algorithm for dominators) is their versatility, that is, they can be run on a W-PRAM with little or no modification. Observe that the inner loop of the start-over algorithm for computing the transitive closure requires O(log n) time as it involves computing a minimum of n elements. This computation can be done in constant time on a W-PRAM [16]. This provides us with an O(log n) time start-over algorithm for computing the transitive closure and an O(1) time algorithm for updating it. In the case of dominators we face a problem with computing ancestor information in constant time. Consequently, updating dominators (which involves updating ancestral information) in O(1) times appears difficult. Our incremental algorithm for updating a topological order uses a known [16] algorithm that merges two sorted list in O(1) time on an R-PRAM. Obviously this algorithm would also require constant time on a W-PRAM. Our incremental algorithms can therefore be adapted to run on a W-PRAM in O(1) time. These are faster by a factor of O(log n) over the star-over algorithms on these two PRAM models.

References

1 A. V. Aho and J. D. Ullman, *"Principles of Compiler Design"*, Addison-Wesley, Reading, Mass., 1977.

[2] A. K. Chandra, "Maximal Parallelism in Matrix Multiplication", RC 6193, IBM Rept., 1975.

[3] G. Cheston, "Incremental Algorithms in Graph Theory", TR 91, Dept. of Computer Science, Univ. of Toronto, 1976.

[4] E. D. Dekel, D. Nassimi and S. Sahni, "Parallel Matrix and Graph Algorithms", *SIAM J. Comp. 10* (1981), pp. 657-675.

[5] S. Even and Y. Shiloach, "An On-line Edge Deletion Problem", *JACM 28* (1982), pp. 1-4.

[6] S. Fujishige, "A Note on the Problem of Updating shortest Paths", *Networks 11* (1981), pp. 317-319.

[7] D. Hirschberg, "Parallel Algorithms for the Transitive Closure and the Connected Component Problem", *Proc. Eighth STOC*, 1976, pp. 55-57.

[8] D. Hirschberg, A. K. Chandra and D. V. Sarwate, "Computing Connected Components on Parallel Computers", *CACM 22* (1979), pp. 461-464.

[9] T. Ibaraki and N. Katoh, "On-line Computation of Transitive Closure of Graphs", *Inf. Proc. Letters 16* (1983), pp. 95-97.

[10] L. Kucera, "Parallel Computation and Conflicts in Memory Access", *Inf. Proc. Letters 14* (1982), pp. 93-96.

[11] S. Pawagi and I. V. Ramakrishnan, "An O(log n) Algorithm for Parallel Update of Minimum Spanning Trees", TR 1452, Dept. of Computer Science, Univ. of Maryland, 1984. Also to appear in Information Processing Letters.

[12] S. Pawagi and I. V. Ramakrishnan, "On Using Multiple Inverted Trees for Parallel Updating of Graph Properties", TR 1502, Dept. of Computer Science,

Univ. of Maryland, 1985. Also to appear in Proc. Allerton Conference.

[13] S. Pawagi and I. V. Ramakrishnan, "Parallel Update of Graph Properties in Logarithmic Time", *Fourteenth International Conference on Parallel Processing*, 1985, pp. 186-193.

[14] C. Savage, "Parallel Algorithms for Some Graph Problems", TR 784, Dept. of Mathematics, Univ. of Illinois, Urbana, 1977.

[15] C. Savage and J. Ja'Ja', "Fast Efficient Parallel Algorithms for Some Graph Problems", *SIAM J. Comp. 10* (1981), pp. 682-691.

[16] Y. Shiloach and U. Vishkin, "Finding the Maximum, Merging and Sorting in a Parallel Computation Model", *J. Algorithms 2* (1981), pp. 88-102.

[17] P. Spira and A. Pan, "On Finding and Updating Spanning Trees and Shortest Paths", *SIAM J. Comp. 4* (1975), pp. 375-380.

[18] Y. Tsin and F. Chin, "Efficient Parallel Algorithms for a Class of Graph-Theoretic Problems", *SIAM J. Comp. 14* (1984), pp. 580-599.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

AD-A162957

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): CAR-TR-148 CS-TR-1551	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR TR 85-1124
6a. NAME OF PERFORMING ORGANIZATION: University of Maryland	6b. OFFICE SYMBOL (If applicable): N/A
7a. NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research	
6c. ADDRESS (City, State and ZIP Code): Center for Automation Research College Park, MD 20742	7b. ADDRESS (City, State and ZIP Code): Bolling Air Force Base Washington, D.C. 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49620-83-C-0082	
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO
61102F	2304	A7	

11. TITLE (Include Security Classification): Updating Properties of Directed Acyclic Graphs on a Parallel Random Access Machine

12. PERSONAL AUTHOR(S): Shaunak Pawagi, I.V. Ramakrishnan

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1985-September	22

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GR	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Fast parallel algorithms are presented for updating the transitive closure, the dominator tree, and a topological ordering of a directed acyclic graph (DAG) when an incremental change has been made to it. The kinds of changes that are considered here include insertion of a vertex or insertion and deletion of an edge. The machine model used is a parallel random access machine which allows simultaneous reads but prohibits simultaneous writes into the same memory location. The algorithms described in this paper require $O(\log n)$ time and use $O(n^3)$ processors. These algorithms are efficient when compared to previously known $O(\log^2 n)$ time algorithms for initial computation of the above mentioned properties of DAGS. We also describe a new algorithm for initial computation of the dominator tree of a DAG. Our algorithm improves the processor complexity of a previously known algorithm [14] by a factor of n, but does not affect the time complexity which remains $O(\log^2 n)$.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Dr. Robert Buchal	(202) 767-4940	NM

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

END

FILMED

2-86

DTIC